The Stanford `libcs106` library, Academic Year 2024-25

```
#include "vector.h"
```

`class Vector<ValueType>`

This class stores an ordered list of values similar to an array. It supports traditional array selection using square brackets, as well as inserting and removing elements. Operations that access elements by index run in O(1) time. Operations, such as insert and remove, that must rearrange elements run in O(N) time.

Constructor

`Vector()`	O(1)	Initializes a new empty vector.
`Vector(n, value)`	O(N)	Initializes a new vector storing *n* copies of the given value.

Methods

`add(value)`	O(1)	Adds a new value to the end of this vector.
`clear()`	O(1)	Removes all elements from this vector.
`equals(vec)`	O(N)	Returns **`true`** if the two vectors contain the same elements in the same order.
`get(index)`	O(1)	Returns the element at the specified index in this vector.
`insert(index, value)`	O(N)	Inserts **`value`** into this vector at the specified index.
`isEmpty()`	O(1)	Returns **`true`** if this vector contains no elements.
`mapAll(fn)`	O(N)	Calls the specified function on each element of this vector in order of ascending index.
`remove(index)`	O(N)	Removes the element at the specified index from this vector.
`set(index, value)`	O(1)	Replaces the element at the specified index in this vector with **`value`**.
`size()`	O(1)	Returns the number of elements in this vector.

sort()	O(NlogN)	Rearranges the elements in this vector into sorted order.
subList(*start*, *length*)	O(N)	Returns a new vector containing elements from a sub-range of this vector.
toString()	O(N)	Returns a printable string representation of this vector.

Operators

for (ValueType elem : vec)	O(N)	Iterates through the elements in a vector in order of ascending index.
***vec*[*index*]**	O(1)	Overloads **[]** to select elements from this vector.
vec1 + vec2	O(N)	Concatenates two vectors.
vec1 += vec2;	O(N)	Adds all of the elements from **vec2** to **vec1**.
vec += value;	O(1)	Adds the single specified value to **vec**.
vec += a, b, c;	O(1)	Adds multiple individual values to **vec**.
vec1 == vec2	O(N)	Returns **true** if **vec1** and **vec2** contain the same elements.
vec1 != vec2	O(N)	Returns **true** if **vec1** and **vec2** are different.
ostream << vec	O(N)	Outputs the contents of the vector to the given output stream.
istream >> vec	O(N)	Reads the contents of the given input stream into the vector.

Constructor detail

```
Vector();
Vector(int n, ValueType value = ValueType());
```

Initializes a new vector. The default constructor creates an empty vector. The second form creates an array with **n** elements, each of which is initialized to **value**; if **value** is missing, the elements are initialized to the default value for the type. You may also provide an initializer list of values. The newly created vector will contain those values in order.

Usage:

```
Vector<ValueType> vec;
Vector<ValueType> vec(n, value);
Vector<ValueType> vec = { value1, value2, value3 };
```

Method detail

`void add(const ValueType& value);`

Adds a new value to the end of this vector.

Usage:

```
vec.add(value);
```

`void clear();`

Removes all elements from this vector.

Usage:

```
vec.clear();
```

`bool equals(const Vector& v) const;`

Returns **`true`** if the two vectors contain exactly the same element values in the same order. Identical in behavior to the **`==`** operator.

Usage:

```
if (vec1.equals(vec2)) ...
```

`const ValueType& get(int index) const;`

Returns the element at the specified index in this vector. This method signals an error if the index is not in the array range.

Usage:

```
ValueType val = vec.get(index);
```

```
void insert(int index, const ValueType& value);
```

Inserts a new value into this vector at the specified index. All subsequent elements are shifted one position to the right. This method signals an error if the index is outside the range from 0 up to and including the length of this vector.

Usage:

```
vec.insert(0, value);
```

```
bool isEmpty() const;
```

Returns **true** if this vector contains no elements.

Usage:

```
if (vec.isEmpty()) ...
```

```
void mapAll(std::function<void (const ValueType&)> fn) const;
```

Calls the specified function on each element of this vector in order of ascending index.

Usage:

```
vec.mapAll(fn);
```

```
ValueType remove(int index);
```

Removes the element at the specified index from this vector and returns it. All subsequent elements are shifted one position to the left. This method signals an error if the index is outside the array range.

Usage:

```
ValueType val = vec.remove(index);
```

```
void set(int index, const ValueType& value);
```

Replaces the element at the specified index in this vector with a new value. The previous value at that index is overwritten. This method signals an error if the index is not in the array range.

Usage:

```
vec.set(index, value);
```

```
int size() const;
```

Returns the number of elements in this vector.

Usage:

```
int nElems = vec.size();
```

```
void sort();
```

Rearrange the elements in this vector into sorted order. Elements are compared using the < operator. In sorted order, the minimum element is placed at index 0; the maximum is at index size()-1.

Usage:

```
vec.sort();
```

```
Vector subList(int start, int length) const;
Vector subList(int start) const;
```

Returns a new vector containing elements from a sub-range of this vector. If only one argument is given, the sub-range length is to the end of this vector. For example, the call of subList(2, 4) would return a new vector containing elements 2-5 of the original vector in its indexes 0-3. Throws an error if the range [start, start+length) is not contained within the range [0, size()].

Usage:

```
Vector<ValueType> sub = vec.subList(start, length);
```

```
string toString() const;
```

Returns a printable string representation of this vector, such as `"{value1, value2, value3}"`.

Usage:

```
string str = vec.toString();
```

Operator detail

```
for (ValueType elem : vec)
for (ValueType& elem : vec)
```

The range-based for loop can be used to iterate through the elements in a collection. The iteration accesses vector elements in order of ascending index. An error is signaled if you attempt to add/remove elements from a collection while iterating over it.

Usage:

```
for (ValueType elem : vec) {
        cout << elem << endl;
}
```

```
for (ValueType& elem : vec) { // if reference type, elements are mutable
        elem *= 2;
}
```

```
ValueType& operator[](int index);
const ValueType& operator[](int index) const;
```

Overloads **[]** to select elements from this vector. This extension enables the use of traditional array notation to get or set individual elements. This method signals an error if the index is outside the array range. The file supports two versions of this operator, one for **const** vectors and one for mutable vectors.

Usage:

```
vec[index]
```

```
Vector operator+(const Vector& vec2) const;
```

Concatenates two vectors.

Usage:

```
vec1 + vec2
```

```
Vector& operator+=(const Vector& vec2);
Vector& operator+=(ValueType value);
```

Adds all of the elements from **vec2** (or the single specified value) to **vec1**. As a convenience, the **Vector** package also overloads the comma operator so that it is possible to initialize a vector like this:

```
Vector<int> digits;
digits += 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
```

Usage:

```
vec1 += vec2;
vec1 += value;
```

`ostream& operator<<(const Vector& vec);`

Outputs the contents of **`vec`** to the given output stream. The output is in the form **`{value1, value2, value3}`** where elements are listed in order of ascending index.

Usage:

```
cout << vec << endl;
```

`istream& operator>>(Vector& vec);`

Reads the contents of the given input stream into **`vec`**. Any previous contents of the vector are replaced. The input is expected to be in the form **`{value1, value2, value3}`** where elements are listed in order of ascending index. If unable to read a proper vector from the stream, the operation results in a stream fail state.

Usage:

```
if (infile >> vec) ...
```
